Statistical analysis of simulation-generated time series

Dontje, T.; Lippert, Th.; Petkov, N.; Schilling, K.

Published in:
Default journal

Publication date:
1992

Parallel Computing 18 (1992) 575–588
North-Holland

Practical aspects and experiences

Statistical analysis of simulation-generated time series: Systolic vs. semi-systolic correlation on the Connection Machine *

T. Dontje [a], Th. Lippert [b], N. Petkov [b] and K. Schilling [b]

[a] *Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1214, USA*
[b] *Fachbereich Physik, Universität Wuppertal, Gaussstrasse 20, W-5600 Wuppertal 1, Germany*

Received 15 April 1991
Revised 9 December 1991

Abstract

Dontje, T., Th. Lippert, N. Petkov and K. Schilling, Statistical analysis of simulation-generated time series: Systolic vs. semi-systolic correlation on the Connection Machine, Parallel Computing 18 (1992) 575–588.

Autocorrelation becomes an increasingly important tool to verify improvements in the state of the simulational art in Latice Gauge Theory. Semi-systolic and full-systolic algorithms are presented which are intensively used for correlation computations on the Connection Machine CM-2. The semi-systolic algorithm makes use of an intrinsic, microprogrammed global-add reduction function which is implemented extremely well on the Connection Machine. Nevertheless, the full-systolic correlation algorithm which makes use only of local communication and computation operations turns out to be substantially superior to the semi-systolic scheme whose basic step involves a non-local sum computation that extends over the entire machine.

Keywords. Systolic algorithms; correlation; Connection Machine; global addition.

1. Introduction – physical aspects

Within the context of large scale scientific computing, such as in lattice gauge field theory (LGT), one is typically faced with the task of analysing large data sets from computer simulations in a way similar to experimental physics where one has to deal with large amounts of data out of a detector. In such applications of computational science, the physics quantities are estimated from an ensemble of configurations of the (discretized) 'physical system', as it comes out of the stochastic (Monte-Carlo) updating procedures.

* Work supported in part by Deutsche Forschungsgemeinschaft grant 'Parallel Computers' - Schi 257/1-4.
Correspondence to: Fachbereich Physik, Universität Wuppertal, Gaussstrasse 20, W-5600 Wuppertal 1, Germany.

One of the key issues in the field is, whether the evolution of the simulated system in the computer is rapid enough to provide us with a sufficient number of statistically independent ensemble members to allow for reasonable estimations. Unfortunately, the interesting physics is met in those situations, where the (spatial) correlation length is very large in the discretized systems under study. For increasing system sizes, *local* Monte-Carlo updating procedures tend to become utterly inefficient in their potential to generate statistically independent configurations. At present, big effort is made in the community of numerical field theorists to improve updating algorithms in order to overcome this critical slowing down and achieve reasonable relaxation times of stochastic processes [1]. For this reason, autocorrelation becomes an increasingly important tool to verify improvements in the state of the simulational art.

In practical situations of lattice quantum chromodynamics, where we search for the properties of the fundamental interactions that bind the quarks, the conceptual basic constituents of matter, into hadrons as they are observed in nature [2], one would at present produce data streams of typical lengths between 1k and 100k, from which to select uncorrelated representatives. This is to be considered as the yet exploratory scale of the problem size: Since it is generally believed that realistic results in LGT can only be gained with computers of the Teraflops class, with distributed memory in the many Gigabyte range, i.e. with the parallel computers of the nineties, we will address here a problem that will stay with us for quite a while, namely how to map time-series studies of Monte Carlo data efficiently onto a massively-parallel machine with distributed memory. Our test case here is the implementation of the correlation on the Connection Machine [1] CM-2.

2. The computational problem

2.1 Correlation

The *correlation* of a sequence $\mathbf{a} = (\ldots, a_0, a_1, a_2, \ldots)$ with a sequence $\mathbf{x} = (\ldots, x_0, x_1, x_2, \ldots)$ is a sequence $\mathbf{y} = (\ldots, y_0, y_1, y_2, \ldots)$ which is computed as follows:

$$y_i := \sum_{j=-\infty}^{\infty} a_j x_{i+j}, \quad i = \ldots, 0, 1, 2, \ldots . \tag{1}$$

The term 'correlation' refers to both the operation and the result of the operation, as well. The *convolution* is another basic operation with sequences [3,4,9]. Since the substitution $a'_j = a_{-j}$ transforms the convolution of $\mathbf{a}$ with $\mathbf{x}$ into a correlation of $\mathbf{a}'$ with $\mathbf{x}$, we consider only the correlation in the following.

As given above, the correlation refers to infinite sequences. In practice, one has to do with finite sequences and in the following we are concerned with computing the correlation of two equally long sequences $\mathbf{a} = (a_0, a_1, \ldots a_{n-1})$ and $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ each of n elements:

$$y_i := \sum_{j=0}^{n-1} a_j x_{i+j}, \quad i = \ldots, 0, 1, 2, \ldots . \tag{2}$$

The *autocorrelation* is an important special case in which both input sequences coincide ($\mathbf{a} \equiv \mathbf{x}$). Note that while the input sequences have n elements each, the result sequence has $2n - 1$ elements.

[1] The Connection Machine is a registered trade mark of Thinking Machines Corporation.

2.2 Systolic implementation

Systolic automata are models of parallel computing structures in which data processing and transfer are pipelined and in which the processing units realize functions of equal load between consecutive communication events. Systolic algorithms are algorithms which, as far as an abstract automaton model is concerned, make use of such structures. For more precise definitions of systolic arrays and algorithms and for many examples, the reader is referred to the monographs under references [3] and [4].

Previous research on systolic algorithms and arrays for convolution and correlation has played an important role in the development of the systolic array concept. In particular, the paper 'Why systolic architectures' [5] by Kung which uses the convolution as an illustration example has undoubtedly contributed a lot to the dissemination of the notion of a systolic array. Subsequently, the convolution and the correlation have been considered as standard applications for testing new theories for the design of systolic algorithms.

Previous research concentrated on the convolution of a long, possibly infinite sequence **x** with a short sequence **a**. This is due to the target applications of this research, the *FIR* filters, where an input signal (the sequence **x**) is convolved with a coefficient set **a** to obtain the filtered output signal (the sequence **y**).

In the applications of lattice gauge theory, one is rather interested in the autocorrelation (**a** ≡ **x**) of time series from computer simulations in postprocessing. To be more specific, we can consider the whole series to be *resident* in the computer memory. This feature of the application is important for choosing an appropriate parallel systolic algorithm among a number of alternatives.

2.3 Systolic algorithms on the Connection Machine

The original motivation for proposing the systolic array concept was the potential which such structures hold for a VLSI implementation [5,6]. The development of the concept in the past decade showed, however, another trend: Most of the systolic algorithms originally proposed for VLSI implementation have actually never been realized as VLSI chips, but rather as highly efficient algorithms for appropriate parallel computers. Features of the systolic algorithms such as balanced load, coordinated movement of data preventing memory access conflicts, pipelined of data processing and transfer, etc., make this type of parallel algorithms very suitable for implementation on distributed memory parallel computers [4]. In order to avoid communication bottlenecks and achieve high efficiency on such computers, it is important to coordinate the movement of data in the given interconnection network. The potential of systolic algorithms for parallel computer implementation has been recognized by leading designers and producers of massively parallel computers at a rather early stage [7].

We use the Connection Machine CM-2 installed at the University of Wuppertal as a massively parallel computer for large scale scientific computing applications. For the problem (correlation) considered in this paper, the following features of the machine are of particular interest:

(i) The Connection Machine involves a large ammount of processing elements. An 8k CM-2, for instance, comprises 8192 bit processors and 256 floating-point units. In the slice-wise mode of exploitation which is relevant for scientific computations, each floating-point unit realizes (at least) four processing nodes so that an 8k CM-2 appears as a parallel computer of (at least) 1024 processing nodes. (In the same mode, a 64k CM-2 appears as a parallel computer of at least 8192 processing nodes.) Note that the parallelism provided by the Connection Machine is in the same order of magnitude as the lengths of the simulation-generated time sequences we want to analyse (see Introduction).

(ii) The hypercube-type interconnection network allows to realize a number of different communication patterns. In particular, the CM-2 can be configured as a linear processor array in which each processing element is connected to its left and right neighbours. This interconnection pattern allows fast communication, since it is realized via direct physical links between the processing elements.

(iii) Microprogrammed routines allow for fast global reduction operations such as summing up data distributed among the processing elements.

3. A semi-systolic algorithm – canonical implementation of the correlation

Equation 2 can be given a structural counterpart in the form of an abstract automaton shown in *Fig. 1*. The small black boxes represent delay elements, and the triangles stand for multipliers. Each multiplier carries out a multiplication of the value laid on its input with the value which is assigned to it. *Fig. 1(a)* illustrates the state of the system at time $t = 0$. The element x_0 of the input sequence **x** is laid on the input of the leftmost multiplier where it is multiplied with the element a_0 from the sequence **a**. In general, the product of x_i with a_i

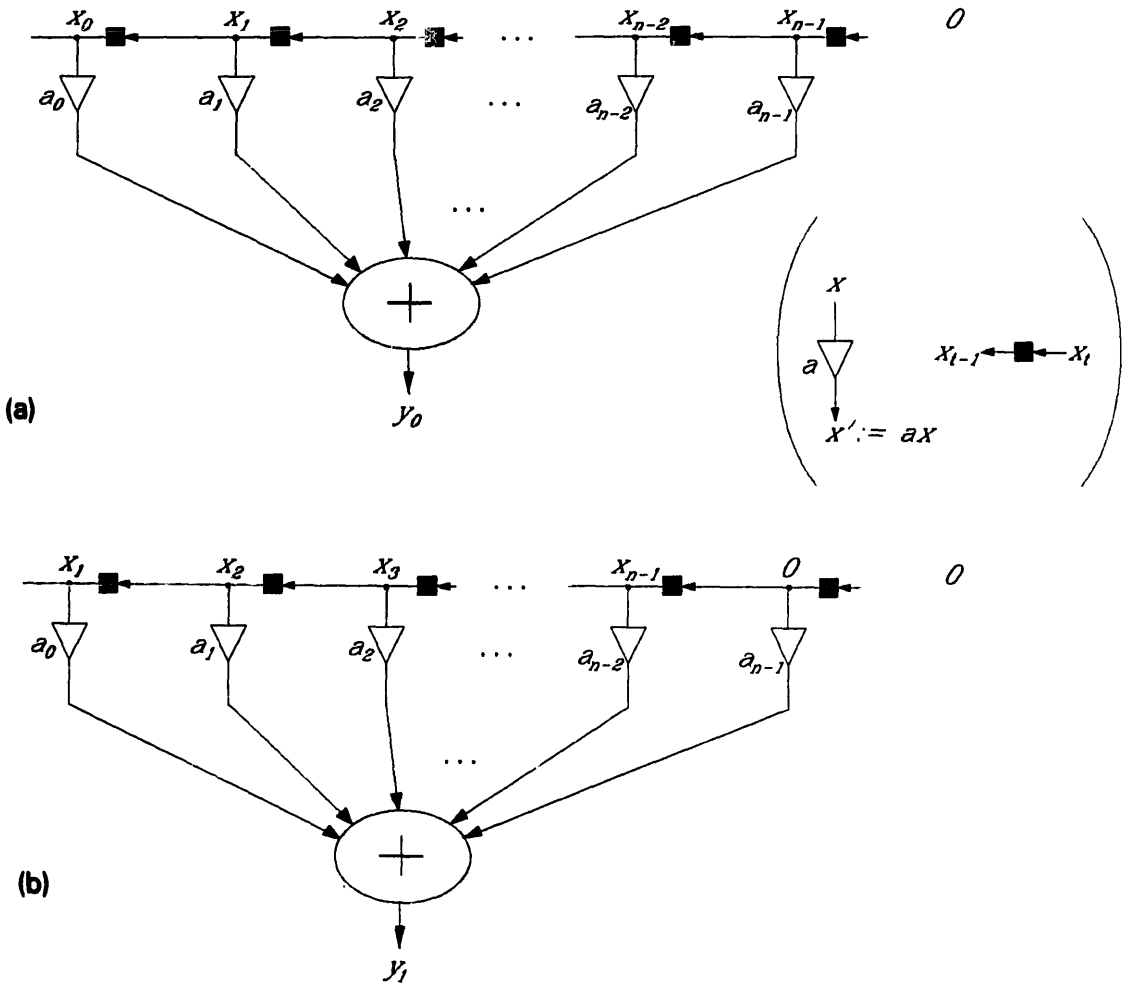

Fig. 1. A structural counterpart of Eq. 2 for the correlation of two sequences: (a) $t = 0$; (b) $t = 1$.

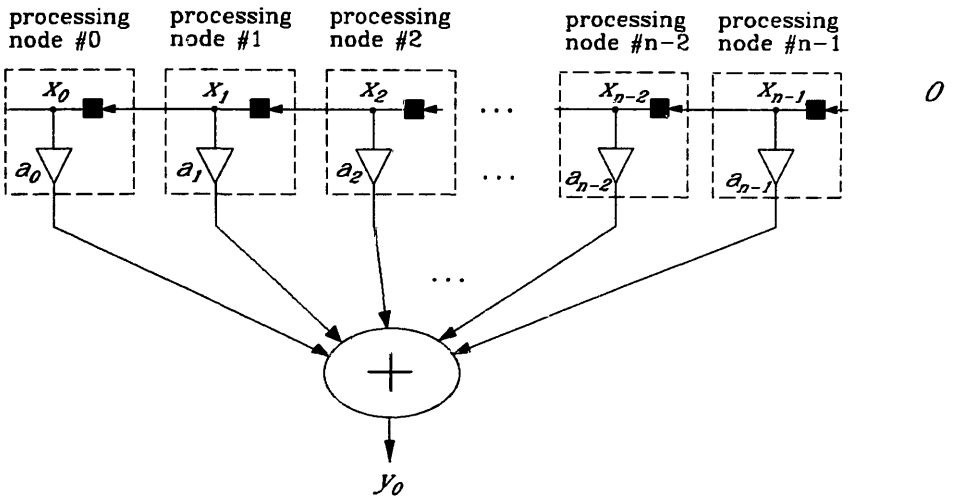

Fig. 2. Mapping an abstract automation structure onto a parallel computer.

($i = 0, \ldots, n-1$) is computed in the ith multiplier in the instant of time illustrated by *Fig. 1(a)*. The products of all mutiplications are forwarded to an adder with n inputs where they are summed together giving the element y_0 of the correlation sequence **y**.

The delay elements are elementary synchronous automata whose function is quite simple: the value laid at the input of a delay element at time $t-1$ appears on its output at time t. Consequently, at time $t = 1$ the elements of the sequence **x** will occupy the positions shown in *Fig. 1(b)*, i.e. the sequence **x** will be shifted by one position to the left. The product $a_i x_{i+1}$ will be computed in the ith multiplier, $i = 0, 1, \ldots, n-1$, and all products will be summed together giving the element y_1 of the correlation **y** according to Eq. 2. In the next instant of time, $t = 2$, y_2 will be computed by this structure followed by y_3 at $t = 3$, etc.

This form of representation of computations by synchronous automata is customary for digital signal processing [9] and is quite useful for the development and analysis of parallel algorithms [3,4]. (In digital signal processing, the particular realization of the correlation by the structure shown in *Fig. 1* is referred to as canonical form.) Its mapping onto an appropriate parallel computer presents no problems, *Fig. 2*: The delay elements, for instance, can be directly mapped onto registers or memory locations in the respective processing nodes, and the functional elements, in this particular case the multipliers, specify the computations to be realized by the processing nodes in each step of the parallel algorithm. Thus, each of the processing nodes shown as dashed-line squares in *Fig. 2* takes two numbers, multiplies them and outputs the product to an adder processing node. After this computation, each processing node outputs one of the multiplicands to its neighbour to the left and replaces it with the number coming from the neighbour on the right. Since the x data is moved to the left, zeros have to be inserted on the right side. These activities constitute one step of the algorithm and this step is repeatedly performed until the last element of the sequence **x** leaves the system on the left.

The structure considered above maps very well onto the Connection Machine: n processing elements of the CM are used. In each of them, two memory locations are reserved, one for an element of the sequence **a** and one for an element of the sequence **x**. In each step of the algorithm, each processing element computes the product of the two values which reside in it and the products computed in the different processing elements are summed together. Although there is no actual global n-input adder in the Connection Machine, the Machine looks to the user like if there were one, since the global addition is specified by one single

microprogrammed intrinsic instruction and is, as we shall see below, rather fast. At the end of the step, each processing element passes to the left the x datum it stores and replaces it by the respective x datum which comes from the right neighbour. Zeros are inserted into the right-most processing element. This basic step is carried out n times and each time it is carried out one element of the correlation is delivered by the global summation.

Here is a simple programme expressing this algorithm in CM Fortran which conforms to Fortran 90:

```
    subroutine semi_sys
    parameter (n=2**k)          ! n - sequence length (power of 2)
    real,array(0:n-1)::a,x,p,y
CMF$LAYOUT  a(:news)            ! distributed sequence a
CMF$LAYOUT  x(:news)            ! distributed sequence x
CMF$LAYOUT  p(:news)            ! distr. auxiliary 'product' sequence
CMF$LAYOUT  y(:serial)          ! correlation sequence
    p    = a*x                  ! mult
    y(0) = sum(p)               ! global add
    do i = 1,n-1                ! step counter (global clock)
       x    = eoshift(x,1,1,0)  ! left shift, zeros inserted right
       p    = a*x               ! mult
       y(i) = sum(p)            ! global add (all elements of p)
    enddo
    return
    end
```

Note that the algorithm described above delivers the elements y_0 through y_{n-1} of the correlation. To compute the values y_{-1} through y_{1-n}, one has to start from the same configuration as the one shown in *Fig. 1*, but this time the x data have to be shifted to the right.

The algorithm classifies as semi-systolic: The word 'systolic' in this term refers to the pipelined shifting of the elements of the sequence x (there is no global broadcasting of data). The attribute 'semi' refers to the global fan-in of the products to sum them. As we shall see below, although this operation is rather fast on the Connection Machine, it is nevertheless inferior to completely local operations. As a consequence, a full-systolic algorithm such as the one presented in the following section will turn out to be faster than the semi-systolic algorithm.

4. A full-systolic algorithm

In the example considered above, the synchronized transfer of data was implemented by means of delay elements controlled by a common global clock. At each tick of the clock, the output of such a delay element copies its input. The systolic algorithm to be presented in this section makes use of similar delay elements which are used as models of the memory locations which keep data that is shifted. These delay elements are, however, clocked at half the normal frequency. This means: the output of such a delay element copies its input only at every second tick of the normal clock. (We remember that the clock has nothing to do with the machine clock of the concrete parallel computer, but is rather an abstraction introduced to partition a parallel computation in steps.)

Two types of delay elements are considered (*Fig. 3*): delay elements which are active at the even ticks of the clock (*Fig. 3(a)*) – for reference we call them the 'white' delay elements – and delay elements which are clocked at the odd ticks of the clock (*Fig. 3(b)*) – we refer to

Fig. 3. 'White' and 'black' delay elements as abstractions for memory locations in which data is shifted in and out during the even and odd steps (clock periods), respectively.

them as the 'black' delay elements. Due to this clocking scheme, the data which occupies memory locations represented by the white and black delay elements is moved in the even and odd clock periods (steps), respectively. In the terms of the Connection Machine, we can say that there are two data sets (CM distributed data arrays) which are shifted alternately.

In systolic array design, it is customary to represent the primitive processing functions of a systolic algorithm by so called cell procedures [3,4]. A cell procedure is executed once in each clock period by a cell and a cell corresponds to a processing element. *Fig. 4(a)* shows a specification of the cell procedure to be used in the systolic algorithm which is presented below. We briefly explain the procedure:

Two variables y^e and y^0 reside in the cell, and they are alternately used in the even and odd clock periods, respectively. Further, the cell procedure has two input variables, a and x, and two output variables a' and x'. In each step (abstract clock period) the input variables a and x are multiplied and their product ax is accumulated in the cell-resident variable y^0 if the clock period (i.e. the step) in which the cell procedure is executed is odd, *Fig. 4(b)*. If the cell procedure is executed in an even clock period, the product ax is accumulated in the cellresident variable y^e, *Fig. 4(c)*. (In each of the cases the respective accumulating variable is underlined.) In both cases, the values of the input variables a and x are also forwarded to the respective output variables a' and x'.

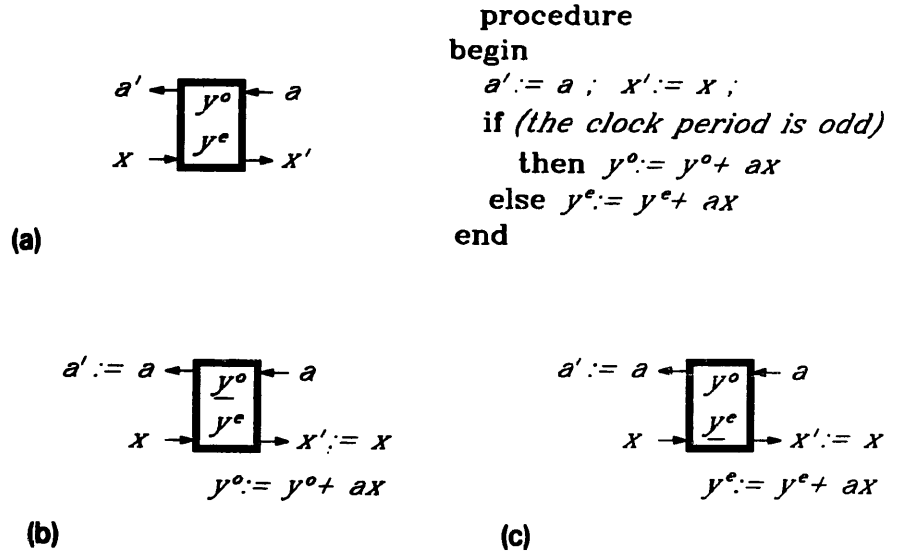

Fig. 4. Cell procedure executed once by a processing node in each step of the algorithm: (a) specification; (b) effect for odd t; (c) effect for even t.

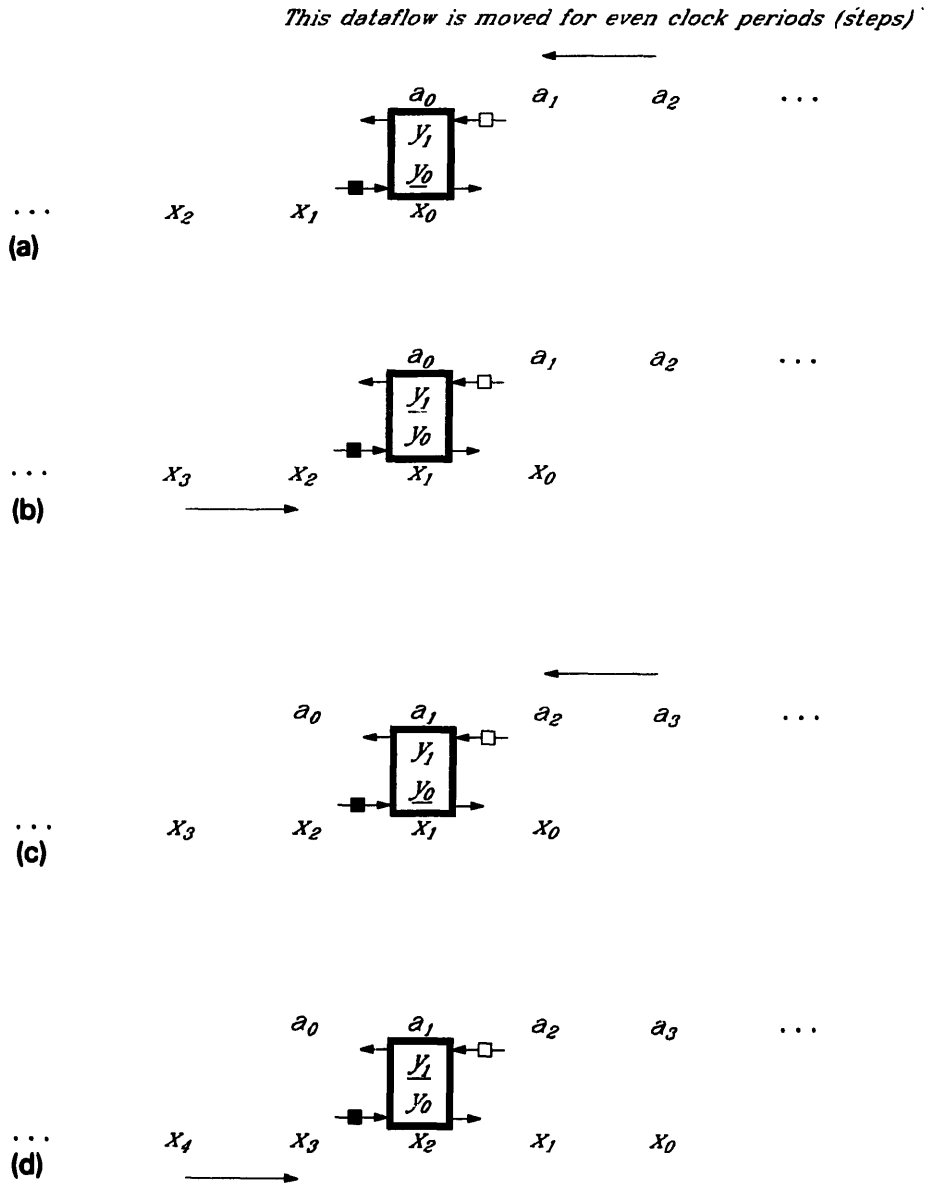

Fig. 5. Computing y_0 and y_1 in one cell (processor). The upper and lower dataflows are moved in the even and odd clock periods (steps), respectively. (a) $t = 0$; (b) $t = 1$ (this dataflow is moved for odd clock periods (steps)); (c) $t = 2$; (d) $t = 3$.

We consider now a cell with one white and one black delay element arranged in its x and a inputs, respectively (*Fig. 5*). The internal variables which are used during even and odd clock periods are denoted by y_0 and y_1, respectively, since when the sequences **a** and **x** are fed into this cell the values of the correlation elements y_0 and y_1 will be accumulated in these cell-resident variables (memory locations). The input operations are represented graphically, this means, the clock period in which a particular data item is inserted into the cell is specified by the distance of this data item to the respective input. Note that the x data is moved (to the right) only for odd ticks of the clock and the a data is moved (to the left) only for even ticks.

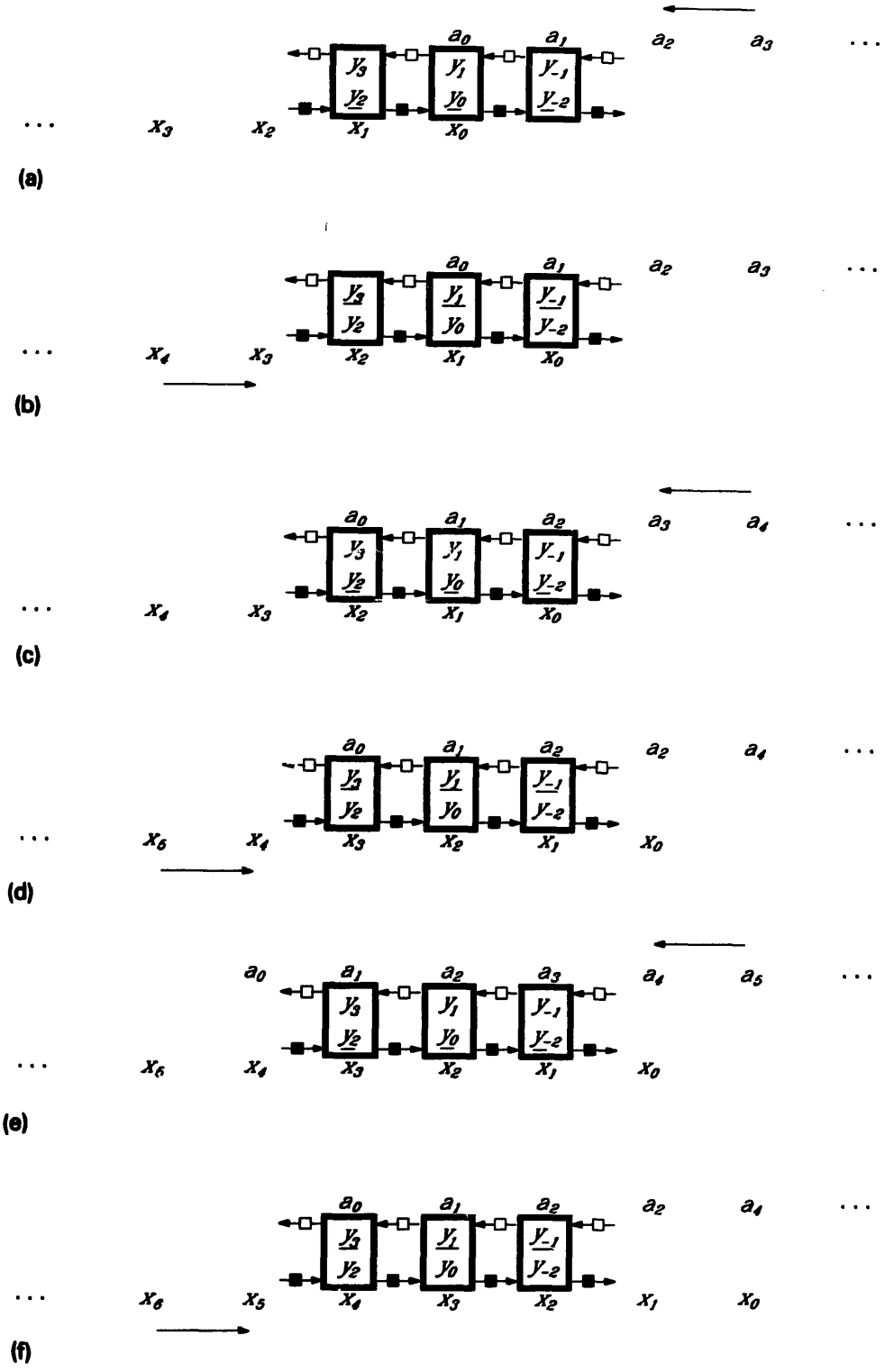

Fig. 6. Dataflow and computational activities; the active/accumulating cell-resident variables are underlined. (a) $t = 0$; (b) $t = 1$; (c) $t = 2$; (d) $t = 3$; (e) $t = 4$; (f) $t = 5$.

We consider the work of the cell step-wise: At $t = 0$, a_0 and x_0 meet in the cell where they are multiplied and accumulated in y_0, *Fig. 5(a)* (the active variable is underlined). In the next clock period, $t = 1$, the x data is shifted to the right, so that now a_0 and x_1 meet in the cell and are multiplied, *Fig. 5(b)*. Since the clock period is odd, the product $a_0 x_1$ is accumulated in y_1. At $t = 2$, the a data is moved to the left, so that now a_1 and x_1 meet in the cell where they are multiplied and their product is accumulated in y_0, *Fig. 5(c)*, At $t = 3$, the x data is moved once again, so that a_1 and x_2 meet in the cell, are multiplied and their product is accumulated to the active cell-resident variable in which y_1 is being accumulated, *Fig. 5(d)*. The input of data proceeds similarly with shifting the a and x data in alternate clock periods until the whole sequences are passed through the cell. By this time the complete values of the components y_0 and y_1 of the correlation will be accumulated in the respective variables in the cell.

This scheme can be mapped directly onto a processing node of a parallel computer. The cell-resident variables and the delay elements arranged in the inputs of the cell are mapped onto memory locations. Four memory locations (per processor) are needed in this case: two for the cell-resident variables and two for the data which is shifted. The cell function is repeatedly executed by the processing node in the two options described above. The dataflow represented so far as being injected from outside comes actually from neighbouring processing elements (not shown in *Fig. 5*).

Figure 6 shows three neighbouring cells with the distribution of the input sequences and their movement for several consecutive clock periods. As illustrated by this figure, while y_0 and y_1 are accumulated in the middle cell, y_2 and y_3 are computed in the left cell and y_{-1} and y_{-2} are computed in the right cell.

In general n cells are required to compute all $2n - 1$ components of the correlation of two sequences of n elements each. A total of $2n$ clock periods is needed: this is the time which elapses between the first computation, the multiplication a_0 and x_0 in the middle cell, and the last computation, the multiplication of a_{n-1} and x_{n-1} in the same cell. The elements of the correlation sequence reside pair-wise in the cells with y_0 and y_1 in the middle cell, and the elements with positive and negative subscripts to the left and right of this cell, respectively. *Figure 7* shows schematically the initial distribution of the input sequences. Note that only the first half of each sequence is resident in the processor array. This implies the insertion of data at both array ends during the execution of the algorithm.

In practice, it is more convenient to start with a distribution of data which is shown in *Fig. 8*. The sequences **a** and **x** are distributed element-wise in the linearly configured processor array. The sequence **a** is stored left to right and the sequence **x** right to left. The algorithm consists of two phases, each of n clock periods. In the first n clock periods the **a** sequence is shifted to the left (only for the odd clock periods) and the sequence **x** is shifted to the right

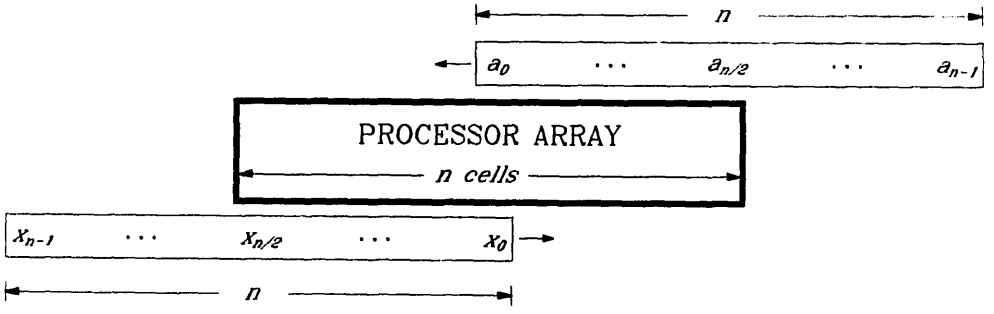

Fig. 7. The straightforward implementation of the systolic algorithm requires insertion of data at both array ends.

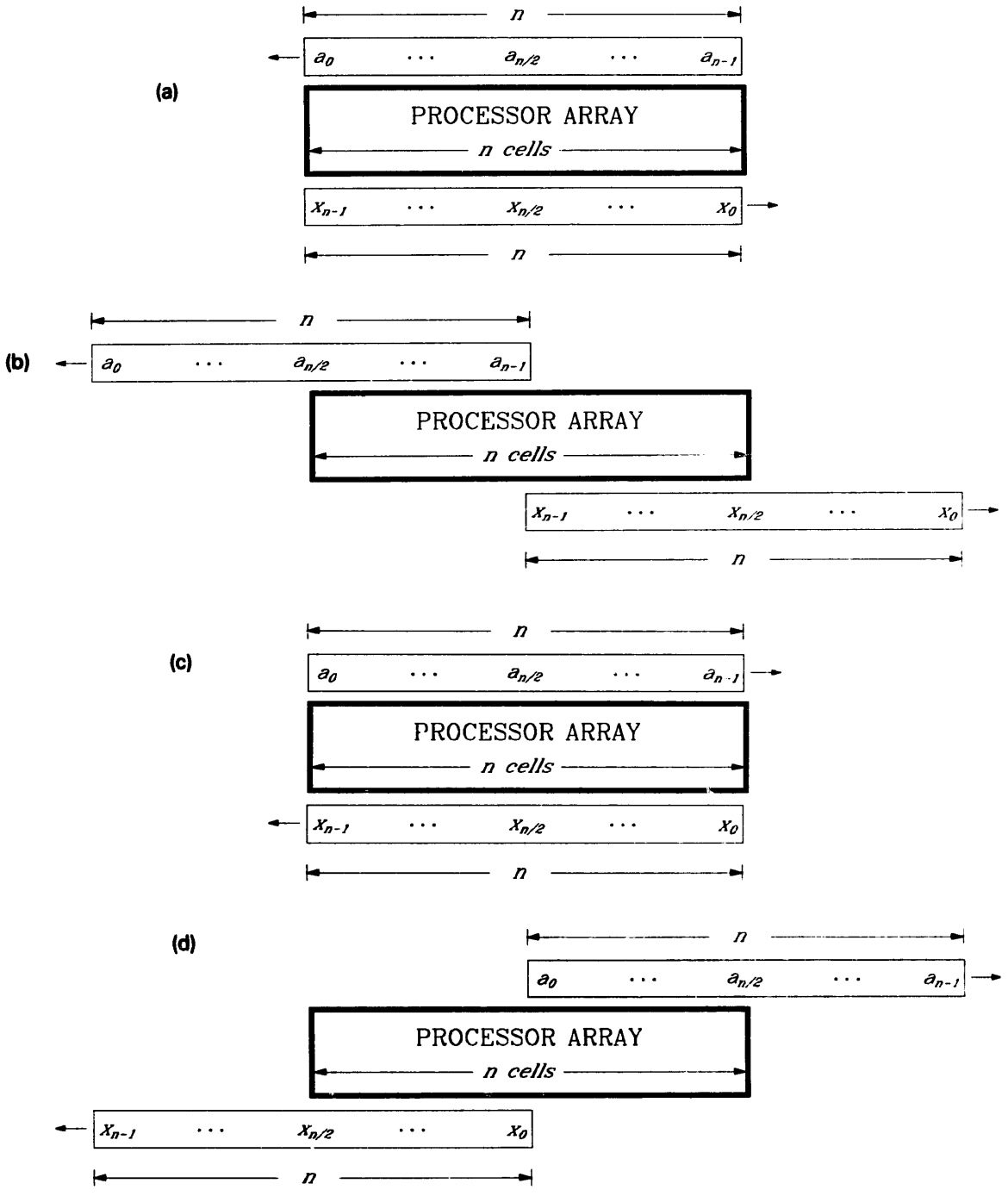

Fig. 8. Data layout and movement for the modified (actually used) algorithm: (a) at the beginning of the first phase; (b) at the end of the first phase (after n clock periods/steps); (c) at the beginning of the second phase; (d) at the end of the second phase.

(only for the even clock periods). Note that after this phase is completed the partial results accumulated in the cell-resident variables for the respective values of the correlation sequence elements correspond only to the correlation of the second halves of the sequences **a** and **x**. In the second phase, the initial distribution of the elements of **a** and **x** is restored

without changing the values of the cell-resident variables and the sequences **a** and **x** are shifted once again for n clock periods but this time in the opposite directions. Here is a CM Fortran expression of the first phase of this algorithm:

```
    subroutine full_sys_first_phase
    parameter(n=2**k)          ! n - sequence length (power of 2)
    real,array(0:n-1)::a,x,ye,yo
CMF$LAYOUT a  (:news)          ! sequence a, arranged as in fig. 8a
CMF$LAYOUT x  (:news)          ! sequence x, arranged as in fig. 8a
CMF$LAYOUT ye (:news)          ! even part of correlation sequence
CMF$LAYOUT yo (:news)          ! odd part of correlation sequence
    yo = 0.
    ye = a*x                   ! even mult
    do i = 1,n/2-1              ! step counter (two clock periods)
       x  = eoshift(x,1,-1,0) ! odd shift (right), zeros inserted
       yo = yo+a*x             ! odd mult-add
       a  = eoshift(a,1,1,0)  ! even shift (left), zeros inserted
       ye = ye+a*x             ! even mult-add
    enddo
    return
    end
```

The second phase differs only in the direction of the shifts.

Since all data movement is pipelined, the algorithm is classified as a full-systolic algorithm. Note that no global communication is required and that each cell communicates merely with its direct neighbours.

5. Results and conclusions

5.1 Complexity considerations and real-life implementation results

Both algorithms given above require an array of n processing elements. Further, both of them need $2n - 1$ clock periods (steps) to compute all $2n - 1$ elements of the correlation of two sequences of n elements. The number of steps is the same for both algorithms but it is not the proper basis for the comparison of the running times of the algorithms, since different operations are realized within these steps. We next consider this in more detail:

In the second, full-systolic algorithm, each processor carries out a shift operation followed by a mult-add operation (multiplication with accumulation) in each step. These operations are completely local and require constant time $\Theta(1)$ which does not depend on the number n of elements in the sequences **x** and **a**. Since the algorithm requires $\Theta(n)$ steps each of time $\Theta(1)$, the overall time requirements of this algorithm are $\Theta(n)$.

In the first, semi-systolic algorithm, each processor carries out one shift operation followed by a multiplication operation in each step. These operations are local and require time $\Theta(1)$ as in the full-systolic algorithm. However, each step involves a global addition operation as well, in which the products computed in the different processors are summed together. The n-input global adder shown in *Figs. 1* and *2* does not actually exist. The global addition is realized by the processors in a sequence of pair-wise additions. The lower bound of the time for the pair-wise addition of n numbers is $\Omega(\log n)$. This lower bound can be reached by a parallel computer with hypercube communications by scanning the levels of a binary addition tree and that is how the global addition is actually done on the Connection Machine [7]. Thus the time required for each step of the first, semi-systolic algorithm is $\Theta(\log n)$. The total running time of this algorithm is therefore $\Theta(n \log n)$.

Table 1
Results measured for $n = 1k$ on an $8k$ Connection Machine CM-2 with a slice-wise compiler and double precision floating-point arithmetics

Semi-systolic algorithm		Full-systolic algorithm	
Operation	Time [μsec]	Operation	Time [μsec]
Shift	124	Shift	124
Mult	18	Multadd	24
Global add	312		
Total step time	454	Total step time	148

According to the above considerations it can be expected that the full-systolic algorithm is faster than the semi-systolic one. However, complexity results imply only asymptotic statements. In our case, they mean that for sufficiently large values of n the full-systolic algorithm should be superior to the semi-systolic one. Anyone who has implemented parallel algorithms on real parallel machines is aware of the fact that in most of the practical cases the asymptotic behaviour of performance measures can be hidden by the constants. Often, an algorithm which is optimal in the complexity sense turns out to be inferior to a suboptimal algorithm in actual problems.

This discrepancy between theory and practice is, however, not the case in our studies of the parallel correlation on the Connection Machine. *Table 1* gives the times of the operations involved in each step of the two algorithms presented above:
As shown in *Table 1*, the step time of the semi-systolic algorithm is nearly three times the step time of the full-systolic algorithm. Since the same number of steps is involved in both algorithms, the overall running times relate in the same way. Note that the superiority of the systolic algorithm would have been even more convincing (by a factor of 15!), if there were no communication overhead for the shift [2] operations.

5.2 Parameter issues

Often, a relatively small number s of elements of the correlation has to be computed. Further, the number p of physical processors which are available will typically be different from the number n of elements in the input sequences. We briefly comment on how the results presented above depend on the values of these parameters.

For $n \leq p$, the step time of the semi-systolic algorithm is $\Theta(\log\ p)$. The running time of the semi-systolic algorithm is therefore $\Theta(s\ \log\ p)$ for s steps needed to compute s successive elements of the correlation. The full-systolic computation of one element of the correlation is substantially a sequential computation and requires $\Theta(n)$ steps to compute and accumulate all products required. (The parallelization in this case refers to the concurrent computation of different elements of the correlation.) Therefore, the running time of the full-systolic algorithm is $\Theta(n)$. We thus have running time $\Theta(\text{s log } p)$ for the semi-systolic algorithm versus $\Theta(n)$ for the full-systolic algorithm ($\text{s} \leq n \leq p$). Which algorithm is superior, depends substantially on the relation between s and n. For relatively small values of s, the semi-systolic algorithm is faster. The full-systolic algorithm is superior for large values of s. The cross-over point which is machine specific lies at about $s = n/3$.For $n > p$, the difference in the step times of both algorithms decreases and vanishes for $n \gg p$. This is due to the fact

[2] The EOSHIFT operation used in the CM Fortran programme specification requires considerably more time. In practice we realise the eoshifts by fast circular shift (CSHIFT) operations and zero assignments in the boundary processors.

that each physical processor simulates many virtual processors and one global addition has to be done only after n/p serial local multiplications and additions. The step time, therefore, is $\Theta(n/p) + \Theta(\log\ p)$ for the semi-systolic and $\Theta(n/p)$ for the full systolic algorithm. For large values of the ratio n/p the term $\Theta(\log\ p)$ becomes negligible in comparison to the term $\Theta(n/p)$. As far as the dependence on s is concerned, the semi-systolic algorithm should be used for $s \ll n$, similar to the previous case.

5.3 Concluding remarks

Although the global add is a reduction function which is implemented extremely well on the Connection Machine, full-systolic correlation/convolution algorithms which make use only of local communication and computation operations turn out to be substantially superior to the semi-systolic scheme whose basic step involves a non-local sum computation that extends over the entire machine. The case example studied in this paper can therefore be considered as an illustration of the usefulness of algorithmic research. The code is being used intensively in a study of global update methods to fight the critical slowing down in U(1) gauge theory close to the phase transition [10–12].

References

[1] U. Wolff, Critical slowing down, in N. Cabibbo et al., eds., *Lattice '89, Proc. 1989 Symp. on Lattice Field Theory*, Capri (1989) (North-Holland, Amsterdam, 1990) 93–102.
[2] C. Rebbi, ed., *Lattice Gauge Theories and Monte Carlo Simulations*, (World Scientific Publ., Singapore, 1983).
[3] N. Petkov, *Systolische Algorithmen und Arrays*, (Akademie-Verlag, Berlin, 1989).
[4] N. Petkov, *Systolic Parallel Processing*, (North-Holland, Amsterdam, to appear).
[5] H.T. Kung, Why systolic architectures, *Computer* 15 (1) (1982) 37–46.
[6] H.T. Kung and C.E. Leiserson, Systolic arrays (for VLSI), *Sparse Matrix Proc. 1978* (Society for Industrial and Applied Mathematics, 1979), 256–282; the same as 'Algorithms for VLSI processor arrays', in C. Mead and L. Conway, *Introduction to VLSI Systems* (Addison-Wesley, Reading, MA, 1980) Sect. 8.3.
[7] D. Hillis and G.L. Steele, Jr., Data parallel algorithms, *Comm. ACM* 29 (12) (1986) 1170–1183.
[8] Thinking Machine Corporation; The Connection Machine CM-2, *Technical Manuals* (TMC, Boston, MA, 1990).
[9] L.R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing* (Prentice-Hall, Englewood Cliffs, NJ, 1975).
[10] S. Adler, G. Bhanot, Th. Lippert, K. Schilling and P. Ueberholz, Fourier acceleration methods in U(1) gauge theory, preprint WUB 91-24, Dept. of Theoretical Physics, Univ. of Wuppertal.
[11] S. Adler, G. Bhanot, Th. Lippert, K. Schilling and P. Ueberholz, Defeating critical slowing down for Abelean gauge dynamics, preprint WUB 91-22, Dept. of Theoretical Physics, Univ. of Wuppertal, IASSNS-HEP-91/39.
[12] S.L. Adler, G. Bhanot, T. Lippert, N. Petkov, K. Schilling and P. Ueberholz, Accelerating Lattice Gauge Computations on the Connection Machine CM-2, *Proc. Parallel Computing '91*, London (Sep. 3-6, 1991) (North-Holland, Amsterdam, to appear 1992).